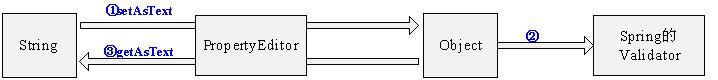
**7.1、简介**

在编写可视化界面项目时，我们通常需要对数据进行类型转换、验证及格式化。

**一、在Spring3之前，我们使用如下架构进行类型转换、验证及格式化：**

  
 **流程：**

①：类型转换：首先调用PropertyEditor的setAsText（String），内部根据需要调用setValue(Object)方法进行设置转换后的值；

②：数据验证：需要显示调用Spring的Validator接口实现进行数据验证；

③：格式化显示：需要调用PropertyEditor的getText进行格式化显示。

**使用如上架构的缺点是：**

（1、PropertyEditor被设计为只能String<——>Object之间转换，不能任意对象类型<——>任意类型，如我们常见的Long时间戳到Date类型的转换是办不到的；

（2、PropertyEditor是线程不安全的，也就是有状态的，因此每次使用时都需要创建一个，不可重用；

（3、PropertyEditor不是强类型的，setValue（Object）可以接受任意类型，因此需要我们自己判断类型是否兼容；

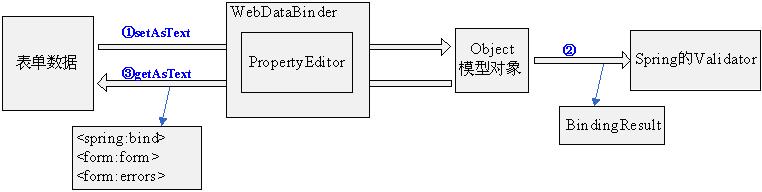
（4、需要自己编程实现验证，Spring3支持更棒的注解验证支持；

（5、在使用SpEL表达式语言或DataBinder时，只能进行String<--->Object之间的类型转换；

（6、不支持细粒度的类型转换/格式化，如UserModel的registerDate需要转换/格式化类似“2012-05-01”的数据，而OrderModel的orderDate需要转换/格式化类似“2012-05-01

15：11：13”的数据，因为大家都为java.util.Date类型，因此不太容易进行细粒度转换/格式化。

**在Spring Web MVC环境中，数据类型转换、验证及格式化通常是这样使用的：**

  
 **流程：**

①、类型转换：首先表单数据（全部是字符串）通过WebDataBinder进行绑定到命令对象，内部通过PropertyEditor实现；

②：数据验证：在控制器中的功能处理方法中，需要显示的调用Spring的Validator实现并将错误信息添加到BindingResult对象中；

③：格式化显示：在表单页面可以通过如下方式展示通过PropertyEditor格式化的数据和错误信息：

**Java代码  [收藏代码](javascript:void())**

1. <%@taglib prefix="spring" uri="http://www.springframework.org/tags" %>
2. <%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

首先需要通过如上taglib指令引入spring的两个标签库。

**Java代码  [收藏代码](javascript:void())**

1. //1、格式化单个命令/表单对象的值（好像比较麻烦，真心没有好办法）
2. <spring:bind path="dataBinderTest.phoneNumber">${status.value}</spring:bind>

**Java代码  [收藏代码](javascript:void())**

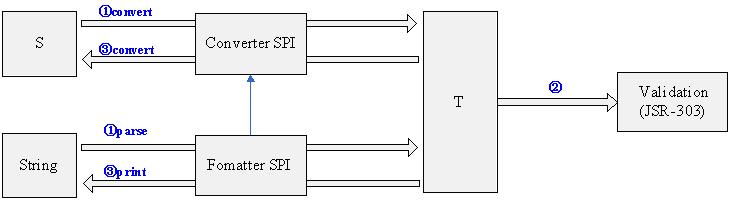
1. //2、通过form标签，内部的表单标签会自动调用命令/表单对象属性对应的PropertyEditor进行格式化显示
2. <form:form commandName="dataBinderTest">
3. <form:input path="phoneNumber"/><!-- 如果出错会显示错误之前的数据而不是空 -->
4. </form:form>

**Java代码  [收藏代码](javascript:void())**

1. //3、显示验证失败后的错误信息
2. <form:errors></form:errors>

如上PropertyEditor和验证API使用起来比较麻烦，而且有许多缺点，因此Spring3提供了更强大的类型转换（Type Conversion）支持，它可以在任意对象之间进行类型转换，不仅仅是String<——>Object；也提供了强大的数据验证支持；同时提供了强大的数据格式化支持。

**二、从Spring3开始，我们可以使用如下架构进行类型转换、验证及格式化：**



**流程：**

①：类型转换：内部的ConversionService会根据S源类型/T目标类型自动选择相应的Converter SPI进行类型转换，而且是强类型的，能在任意类型数据之间进行转换；

②：数据验证：支持JSR-303验证框架，如将@Valid放在需要验证的目标类型上即可；

③：格式化显示：其实就是任意目标类型---->String的转换，完全可以使用Converter SPI完成。

Spring为了更好的诠释格式化/解析功能提供了Formatter SPI，支持根据Locale信息进行格式化/解析，而且该套SPI可以支持字段/参数级别的细粒度格式化/解析，**流程如下：**

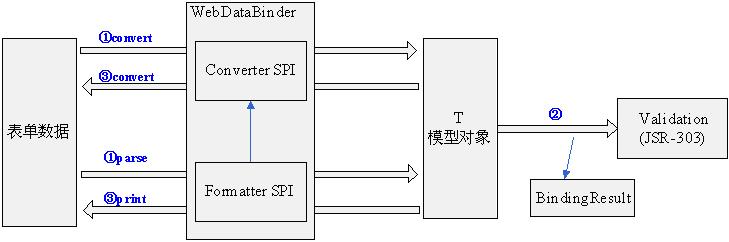
①：类型解析（转换）：String---->T类型目标对象的解析，和PropertyEditor类似；

③：格式化显示：任意目标类型---->String的转换，和PropertyEditor类似。

Formatter SPI最大特点是能进行字段/参数级别的细粒度解析/格式化控制，即使是Converter SPI也是粗粒度的（到某个具体类型，而不是其中的某个字段单独控制），目前Formatter SPI还不是很完善，如果您有好的想法可以到Spring官网提建议。

Formatter SPI内部实现实际委托给Converter SPI进行转换，即约束为解析/格式化String<---->任意目标类型。

**在Spring Web MVC环境中，数据类型转换、验证及格式化通常是这样使用的：**



①、类型转换：首先表单数据（全部是字符串）通过WebDataBinder进行绑定到命令对象，内部通过Converter SPI实现；

②：数据验证：使用JSR-303验证框架进行验证；

③：格式化显示：在表单页面可以通过如下方式展示通过内部通过Converter SPI格式化的数据和错误信息：

**Java代码  [收藏代码](javascript:void())**

1. <%@taglib prefix="spring" uri="http://www.springframework.org/tags" %>
2. <%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

首先需要通过如上taglib指令引入spring的两个标签库。

**Java代码  [收藏代码](javascript:void())**

1. //1、格式化单个命令/表单对象的值（好像比较麻烦，真心没有好办法）
2. <spring:bind path="dataBinderTest.phoneNumber">${status.value}</spring:bind>

**Java代码  [收藏代码](javascript:void())**

1. //2、<spring:eval>标签，自动调用ConversionService并选择相应的Converter SPI进行格式化展示
2. <spring:eval expression="dataBinderTest.phoneNumber"></spring:eval>

如上代码能工作的前提是在RequestMappingHandlerMapping配置了ConversionServiceExposingInterceptor，它的作用是暴露conversionService到请求中以便如<spring:eval>标签使用。

**Java代码  [收藏代码](javascript:void())**

1. //3、通过form标签，内部的表单标签会自动调用命令/表单对象属性对应的PropertyEditor进行格式化显示
2. <form:form commandName="dataBinderTest">
3. <form:input path="phoneNumber"/><!-- 如果出错会显示错误之前的数据而不是空 -->
4. </form:form>

**Java代码  [收藏代码](javascript:void())**

1. //4、显示验证失败后的错误信息
2. <form:errors></form:errors>

接下来我们就详细学习一下这些知识吧。

**7.2、数据类型转换**

**7.2.1、Spring3之前的PropertyEditor**

PropertyEditor介绍请参考【4.16.1、数据类型转换】。

**一、测试之前我们需要准备好测试环境：**

（1、模型对象，和【4.16.1、数据类型转换】使用的一样，需要将DataBinderTestModel模型类及相关类拷贝过来放入cn.javass.chapter7.model包中。

（2、控制器定义：

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter7.web.controller;
2. //省略import
3. @Controller
4. **public** **class** DataBinderTestController {
5. @RequestMapping(value = "/dataBind")
6. **public** String test(DataBinderTestModel command) {
7. //输出command对象看看是否绑定正确
8. System.out.println(command);
9. model.addAttribute("dataBinderTest", command);
10. **return** "bind/success";
11. }
12. }

 (3、Spring配置文件定义，请参考chapter7-servlet.xml，并注册DataBinderTestController：

**Java代码  [收藏代码](javascript:void())**

1. <bean **class**="cn.javass.chapter7.web.controller.DataBinderTestController"/>

（4、测试的URL：

http://localhost:9080/springmvc-chapter7/dataBind?username=zhang&bool=yes&schooInfo.specialty=computer&hobbyList[0]=program&hobbyList[1]=music&map[key1]=value1&map[key2]=value2&phoneNumber=010-12345678&date=2012-3-18 16:48:48&state=blocked

**二、注解式控制器注册PropertyEditor：**

**1、使用WebDataBinder进行控制器级别注册PropertyEditor（控制器独享）**

**Java代码  [收藏代码](javascript:void())**

1. @InitBinder
2. //此处的参数也可以是ServletRequestDataBinder类型
3. **public** **void** initBinder(WebDataBinder binder) **throws** Exception {
4. //注册自定义的属性编辑器
5. //1、日期
6. DateFormat df = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
7. CustomDateEditor dateEditor = **new** CustomDateEditor(df, **true**);
8. //表示如果命令对象有Date类型的属性，将使用该属性编辑器进行类型转换
9. binder.registerCustomEditor(Date.**class**, dateEditor);
10. //自定义的电话号码编辑器(和【4.16.1、数据类型转换】一样)
11. binder.registerCustomEditor(PhoneNumberModel.**class**, **new** PhoneNumberEditor());
12. }

和【4.16.1、数据类型转换】一节类似，只是此处需要通过@InitBinder来注册自定义的PropertyEditor。

**2、使用WebBindingInitializer批量注册PropertyEditor**

和【4.16.1、数据类型转换】不太一样，因为我们的注解式控制器是POJO，没有实现任何东西，因此无法注入WebBindingInitializer，此时我们需要把WebBindingInitializer注入到我们的RequestMappingHandlerAdapter或AnnotationMethodHandlerAdapter，这样对于所有的注解式控制器都是共享的。

**Java代码  [收藏代码](javascript:void())**

1. <bean **class**="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
2. <property name="webBindingInitializer">
3. <bean **class**="cn.javass.chapter7.web.controller.support.initializer.MyWebBindingInitializer"/>
4. </property>
5. </bean>

此时我们注释掉控制器级别通过@InitBinder注册PropertyEditor的方法。

**3、全局级别注册PropertyEditor（全局共享）**

和【4.16.1、数据类型转换】一节一样，此处不再重复。请参考【4.16.1、数据类型转换】的【全局级别注册PropertyEditor（全局共享）】。

接下来我们看一下Spring3提供的更强大的类型转换支持。

**7.2.2、Spring3开始的类型转换系统**

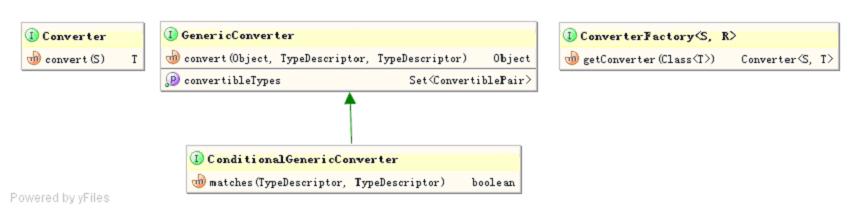
Spring3引入了更加通用的类型转换系统，其定义了SPI接口（Converter等）和相应的运行时执行类型转换的API（ConversionService等），在Spring中它和PropertyEditor功能类似，可以替代PropertyEditor来转换外部Bean属性的值到Bean属性需要的类型。

该类型转换系统是Spring通用的，其定义在org.springframework.core.convert包中，不仅仅在Spring Web MVC场景下。目标是完全替换PropertyEditor，提供无状态、强类型且可以在任意类型之间转换的类型转换系统，可以用于任何需要的地方，如SpEL、数据绑定。

Converter SPI完成通用的类型转换逻辑，如java.util.Date<---->java.lang.Long或java.lang.String---->PhoneNumberModel等。

**7.2.2.1、架构**

**1、类型转换器：**提供类型转换的实现支持。



一个有如下三种接口：

**（1、Converter：**类型转换器，用于转换S类型到T类型，此接口的实现必须是线程安全的且可以被共享。

**Java代码  [收藏代码](javascript:void())**

1. **package** org.springframework.core.convert.converter;
2. **public** **interface** Converter<S, T> { //① S是源类型 T是目标类型
3. T convert(S source); //② 转换S类型的source到T目标类型的转换方法
4. }

示例：请参考cn.javass.chapter7.converter.support.StringToPhoneNumberConverter转换器，用于将String--->PhoneNumberModel。

此处我们可以看到Converter接口实现只能转换一种类型到另一种类型，不能进行多类型转换，如将一个数组转换成集合，如（String[] ----> List<String>、String[]----->List<PhoneNumberModel>等）。

**（2、GenericConverter和ConditionalGenericConverter：**GenericConverter接口实现能在多种类型之间进行转换，ConditionalGenericConverter是有条件的在多种类型之间进行转换。

**Java代码  [收藏代码](javascript:void())**

1. **package** org.springframework.core.convert.converter;
2. **public** **interface** GenericConverter {
3. Set<ConvertiblePair> getConvertibleTypes();
4. Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
5. }

getConvertibleTypes:指定了可以转换的目标类型对；

convert：在sourceType和targetType类型之间进行转换。

**Java代码  [收藏代码](javascript:void())**

1. **package** org.springframework.core.convert.converter;
2. **public** **interface** ConditionalGenericConverter **extends** GenericConverter {
3. **boolean** matches(TypeDescriptor sourceType, TypeDescriptor targetType);
4. }

matches：用于判断sourceType和targetType类型之间能否进行类型转换。

示例：如org.springframework.core.convert.support.ArrayToCollectionConverter和CollectionToArrayConverter用于在数组和集合间进行转换的ConditionalGenericConverter实现，如在String[]<---->List<String>、String[]<---->List<PhoneNumberModel>等之间进行类型转换。

对于我们大部分用户来说一般不需要自定义GenericConverter, 如果需要可以参考内置的GenericConverter来实现自己的。

**（3、ConverterFactory：**工厂模式的实现，用于选择将一种S源类型转换为R类型的子类型T的转换器的工厂接口。

**Java代码  [收藏代码](javascript:void())**

1. **package** org.springframework.core.convert.converter;
2. **public** **interface** ConverterFactory<S, R> {
3. <T **extends** R> Converter<S, T> getConverter(Class<T> targetType);
4. }

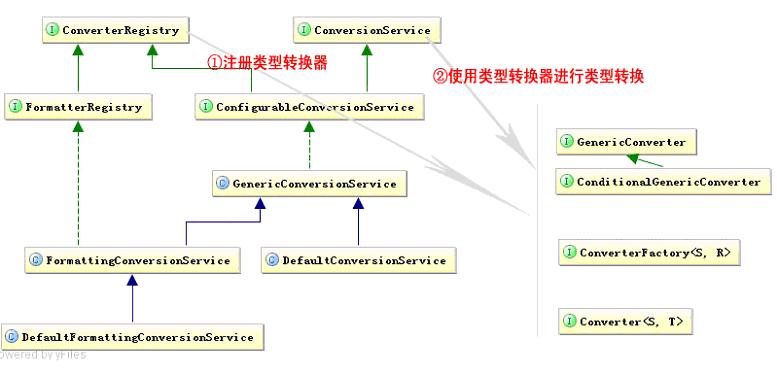
S：源类型；R目标类型的父类型；T：目标类型，且是R类型的子类型；

getConverter：得到目标类型的对应的转换器。

示例：如org.springframework.core.convert.support.NumberToNumberConverterFactory用于在Number类型子类型之间进行转换，如Integer--->Double， Byte---->Integer， Float--->Double等。

对于我们大部分用户来说一般不需要自定义ConverterFactory，如果需要可以参考内置的ConverterFactory来实现自己的。

**2、类型转换器注册器、类型转换服务：**提供类型转换器注册支持，运行时类型转换API支持。



一共有如下两种接口：

**（1、ConverterRegistry：**类型转换器注册支持，可以注册/删除相应的类型转换器。

**Java代码  [收藏代码](javascript:void())**

1. **package** org.springframework.core.convert.converter;
2. **public** **interface** ConverterRegistry {
3. **void** addConverter(Converter<?, ?> converter);
4. **void** addConverter(Class<?> sourceType, Class<?> targetType, Converter<?, ?> converter);
5. **void** addConverter(GenericConverter converter);
6. **void** addConverterFactory(ConverterFactory<?, ?> converterFactory);
7. **void** removeConvertible(Class<?> sourceType, Class<?> targetType);
8. }

可以注册：Converter实现，GenericConverter实现，ConverterFactory实现。

**（2、ConversionService：**运行时类型转换服务接口，提供运行期类型转换的支持。

**Java代码  [收藏代码](javascript:void())**

1. **package** org.springframework.core.convert;
2. **public** **interface** ConversionService {
3. **boolean** canConvert(Class<?> sourceType, Class<?> targetType);
4. **boolean** canConvert(TypeDescriptor sourceType, TypeDescriptor targetType);
5. <T> T convert(Object source, Class<T> targetType);
6. Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);
7. }

convert：将源对象转换为目标类型的目标对象。

Spring提供了两个默认实现（其都实现了ConverterRegistry**、**ConversionService接口）:

DefaultConversionService:默认的类型转换服务实现；

DefaultFormattingConversionService：带数据格式化支持的类型转换服务实现，一般使用该服务实现即可。

**7.2.2.2、Spring内建的类型转换器如下所示：**

|  |  |
| --- | --- |
| **类名** | **说明** |
| **第一组：标量转换器** |  |
| StringToBooleanConverter | String----->Boolean  true:true/on/yes/1； false:false/off/no/0 |
| ObjectToStringConverter | Object----->String  调用toString方法转换 |
| StringToNumberConverterFactory | String----->Number（如Integer、Long等） |
| NumberToNumberConverterFactory | Number子类型(Integer、Long、Double等)<——> Number子类型(Integer、Long、Double等) |
| StringToCharacterConverter | String----->java.lang.Character  取字符串第一个字符 |
| NumberToCharacterConverter | Number子类型(Integer、Long、Double等)——> java.lang.Character |
| CharacterToNumberFactory | java.lang.Character ——>Number子类型(Integer、Long、Double等) |
| StringToEnumConverterFactory | String----->enum类型  通过Enum.valueOf将字符串转换为需要的enum类型 |
| EnumToStringConverter | enum类型----->String  返回enum对象的name()值 |
| StringToLocaleConverter | String----->java.util.Local |
| PropertiesToStringConverter | java.util.Properties----->String  默认通过ISO-8859-1解码 |
| StringToPropertiesConverter | String----->java.util.Properties  默认使用ISO-8859-1编码 |
| **第二组：集合、数组相关转换器** |  |
| ArrayToCollectionConverter | 任意S数组---->任意T集合（List、Set） |
| CollectionToArrayConverter | 任意T集合（List、Set）---->任意S数组 |
| ArrayToArrayConverter | 任意S数组<---->任意T数组 |
| CollectionToCollectionConverter | 任意T集合（List、Set）<---->任意T集合（List、Set）  即集合之间的类型转换 |
| MapToMapConverter | Map<---->Map之间的转换 |
| ArrayToStringConverter | 任意S数组---->String类型 |
| StringToArrayConverter | String----->数组  默认通过“,”分割，且去除字符串的两边空格(trim) |
| ArrayToObjectConverter | 任意S数组---->任意Object的转换  (如果目标类型和源类型兼容，直接返回源对象；否则返回S数组的第一个元素并进行类型转换) |
| ObjectToArrayConverter | Object----->单元素数组 |
| CollectionToStringConverter | 任意T集合（List、Set）---->String类型 |
| StringToCollectionConverter | String----->集合（List、Set）  默认通过“,”分割，且去除字符串的两边空格(trim) |
| CollectionToObjectConverter | 任意T集合---->任意Object的转换  (如果目标类型和源类型兼容，直接返回源对象；否则返回S数组的第一个元素并进行类型转换) |
| ObjectToCollectionConverter | Object----->单元素集合 |
| **第三组：默认（fallback）转换器：**之前的转换器不能转换时调用 |  |
| ObjectToObjectConverter | Object（S）----->Object（T）  首先尝试valueOf进行转换、没有则尝试new 构造器(S) |
| IdToEntityConverter | Id(S)----->Entity(T)  查找并调用public static T find[EntityName](S)获取目标对象，EntityName是T类型的简单类型 |
| FallbackObjectToStringConverter | Object----->String  ConversionService作为恢复使用，即其他转换器不能转换时调用（执行对象的toString()方法） |

S：代表源类型，T：代表目标类型

如上的转换器在使用转换服务实现DefaultConversionService和DefaultFormattingConversionService时会自动注册。

**7.2.2.3、示例**

（1、自定义String----->PhoneNumberModel的转换器

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter7.web.controller.support.converter;
2. //省略import
3. **public** **class** StringToPhoneNumberConverter **implements** Converter<String, PhoneNumberModel> {
4. Pattern pattern = Pattern.compile("^(\\d{3,4})-(\\d{7,8})$");
5. @Override
6. **public** PhoneNumberModel convert(String source) {
7. **if**(!StringUtils.hasLength(source)) {
8. //①如果source为空 返回null
9. **return** **null**;
10. }
11. Matcher matcher = pattern.matcher(source);
12. **if**(matcher.matches()) {
13. //②如果匹配 进行转换
14. PhoneNumberModel phoneNumber = **new** PhoneNumberModel();
15. phoneNumber.setAreaCode(matcher.group(1));
16. phoneNumber.setPhoneNumber(matcher.group(2));
17. **return** phoneNumber;
18. } **else** {
19. //③如果不匹配 转换失败
20. **throw** **new** IllegalArgumentException(String.format("类型转换失败，需要格式[010-12345678]，但格式是[%s]", source));
21. }
22. }
23. }

String转换为Date的类型转换器，请参考cn.javass.chapter7.web.controller.support.converter.StringToDateConverter。

(2、测试用例(cn.javass.chapter7.web.controller.support.converter.ConverterTest)

**Java代码  [收藏代码](javascript:void())**

1. @Test
2. **public** **void** testStringToPhoneNumberConvert() {
3. DefaultConversionService conversionService = **new** DefaultConversionService();
4. conversionService.addConverter(**new** StringToPhoneNumberConverter());
6. String phoneNumberStr = "010-12345678";
7. PhoneNumberModel phoneNumber = conversionService.convert(phoneNumberStr, PhoneNumberModel.**class**);
9. Assert.assertEquals("010", phoneNumber.getAreaCode());
10. }

类似于PhoneNumberEditor将字符串“010-12345678”转换为PhoneNumberModel。

**Java代码  [收藏代码](javascript:void())**

1. @Test
2. **public** **void** testOtherConvert() {
3. DefaultConversionService conversionService = **new** DefaultConversionService();
5. //"1"--->true（字符串“1”可以转换为布尔值true）
6. Assert.assertEquals(Boolean.valueOf(**true**), conversionService.convert("1", Boolean.**class**));
8. //"1,2,3,4"--->List（转换完毕的集合大小为4）
9. Assert.assertEquals(4, conversionService.convert("1,2,3,4", List.**class**).size());
10. }

其他类型转换器使用也是类似的，此处不再重复。

**7.2.2.4、集成到Spring Web MVC环境**

（1、注册ConversionService实现和自定义的类型转换器

**Java代码  [收藏代码](javascript:void())**

1. <!-- ①注册ConversionService -->
2. <bean id="conversionService" **class**="org.springframework.format.support.
3. FormattingConversionServiceFactoryBean">
4. <property name="converters">
5. <list>
6. <bean **class**="cn.javass.chapter7.web.controller.support.
7. converter.StringToPhoneNumberConverter"/>
8. <bean **class**="cn.javass.chapter7.web.controller.support.
9. converter.StringToDateConverter">
10. <constructor-arg value="yyyy-MM-dd"/>
11. </bean>
12. </list>
13. </property>
14. </bean>

FormattingConversionServiceFactoryBean：是FactoryBean实现，默认使用DefaultFormattingConversionService转换器服务实现；

converters：注册我们自定义的类型转换器，此处注册了String--->PhoneNumberModel和String--->Date的类型转换器。

（2、通过ConfigurableWebBindingInitializer注册ConversionService

**Java代码  [收藏代码](javascript:void())**

1. <!-- ②使用ConfigurableWebBindingInitializer注册conversionService -->
2. <bean id="webBindingInitializer" **class**="org.springframework.web.bind.support.
3. ConfigurableWebBindingInitializer">
4. <property name="conversionService" ref="conversionService"/>
5. </bean>

此处我们通过ConfigurableWebBindingInitializer绑定初始化器进行ConversionService的注册；

3、注册ConfigurableWebBindingInitializer到RequestMappingHandlerAdapter

**Java代码  [收藏代码](javascript:void())**

1. <bean **class**="org.springframework.web.servlet.mvc.method.annotation.
2. RequestMappingHandlerAdapter">
3. <property name="webBindingInitializer" ref="webBindingInitializer"/>
4. </bean>

通过如上配置，我们就完成了Spring3.0的类型转换系统与Spring Web MVC的集成。此时可以启动服务器输入之前的URL测试了。

此时可能有人会问，如果我同时使用PropertyEditor和ConversionService，执行顺序是什么呢？内部首先查找PropertyEditor进行类型转换，如果没有找到相应的PropertyEditor再通过ConversionService进行转换。

如上集成过程看起来比较麻烦，后边我们会介绍<mvc:annotation-driven>和@EnableWebMvc，ConversionService会自动注册，后续章节再详细介绍。

**7.3、数据格式化**

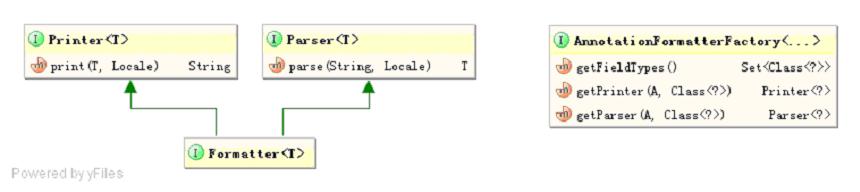
在如Web /客户端项目中，通常需要将数据转换为具有某种格式的字符串进行展示，因此上节我们学习的数据类型转换系统核心作用不是完成这个需求，因此Spring3引入了格式化转换器（Formatter SPI） 和格式化服务API（FormattingConversionService）从而支持这种需求。在Spring中它和PropertyEditor功能类似，可以替代PropertyEditor来进行对象的解析和格式化，而且支持细粒度的字段级别的格式化/解析。

Formatter SPI核心是完成解析和格式化转换逻辑，在如Web应用/客户端项目中，需要解析、打印/展示本地化的对象值时使用，如根据Locale信息将java.util.Date---->java.lang.String打印/展示、java.lang.String---->java.util.Date等。

该格式化转换系统是Spring通用的，其定义在org.springframework.format包中，不仅仅在Spring Web MVC场景下。

**7.3.1、架构**

**1、格式化转换器：**提供格式化转换的实现支持。



一共有如下两组四个接口：

**（1、Printer接口：**格式化显示接口，将T类型的对象根据Locale信息以某种格式进行打印显示（即返回字符串形式）；

**Java代码  [收藏代码](javascript:void())**

1. **package** org.springframework.format;
2. **public** **interface** Printer<T> {
3. String print(T object, Locale locale);
4. }

**（2、Parser接口：**解析接口，根据Locale信息解析字符串到T类型的对象；

**Java代码  [收藏代码](javascript:void())**

1. **package** org.springframework.format;
2. **public** **interface** Parser<T> {
3. T parse(String text, Locale locale) **throws** ParseException;
4. }

解析失败可以抛出java.text.ParseException或IllegalArgumentException异常即可。

**（3、Formatter接口：**格式化SPI接口，继承Printer和Parser接口，完成T类型对象的格式化和解析功能；

**Java代码  [收藏代码](javascript:void())**

1. **package** org.springframework.format;
2. **public** **interface** Formatter<T> **extends** Printer<T>, Parser<T> {
3. }

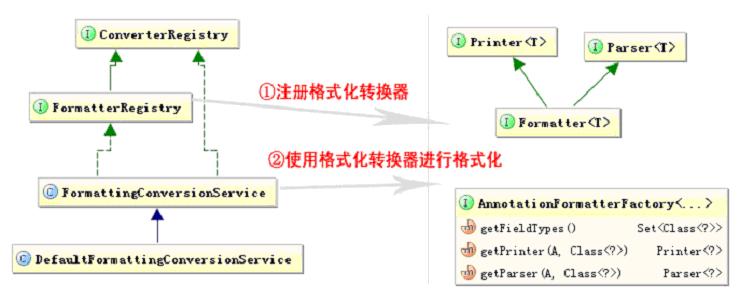
**（4、AnnotationFormatterFactory接口：**注解驱动的字段格式化工厂，用于创建带注解的对象字段的Printer和Parser，即用于格式化和解析带注解的对象字段。

**Java代码  [收藏代码](javascript:void())**

1. **package** org.springframework.format;
2. **public** **interface** AnnotationFormatterFactory<A **extends** Annotation> {//①可以识别的注解类型
3. Set<Class<?>> getFieldTypes();//②可以被A注解类型注解的字段类型集合
4. Printer<?> getPrinter(A annotation, Class<?> fieldType);//③根据A注解类型和fieldType类型获取Printer
5. Parser<?> getParser(A annotation, Class<?> fieldType);//④根据A注解类型和fieldType类型获取Parser
7. }

返回用于格式化和解析被A注解类型注解的字段值的Printer和Parser。如JodaDateTimeFormatAnnotationFormatterFactory可以为带有@DateTimeFormat注解的java.util.Date字段类型创建相应的Printer和Parser进行格式化和解析。

**2、格式化转换器注册器、格式化服务：**提供类型转换器注册支持，运行时类型转换API支持。



一个有如下两种接口：

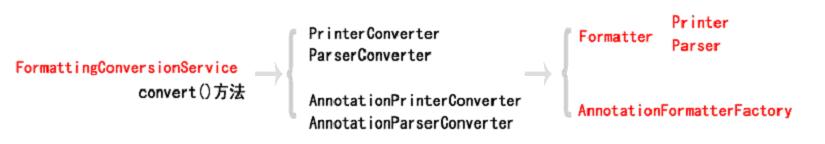
（1、FormatterRegistry：格式化转换器注册器，用于注册格式化转换器（Formatter、Printer和Parser、AnnotationFormatterFactory）；

**Java代码  [收藏代码](javascript:void())**

1. **package** org.springframework.format;
2. **public** **interface** FormatterRegistry **extends** ConverterRegistry {
3. //①添加格式化转换器（Spring3.1 新增API）
4. **void** addFormatter(Formatter<?> formatter);
5. //②为指定的字段类型添加格式化转换器
6. **void** addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter);
7. //③为指定的字段类型添加Printer和Parser
8. **void** addFormatterForFieldType(Class<?> fieldType, Printer<?> printer, Parser<?> parser);
9. //④添加注解驱动的字段格式化工厂AnnotationFormatterFactory
10. **void** addFormatterForFieldAnnotation(
11. AnnotationFormatterFactory<? **extends** Annotation> annotationFormatterFactory);
12. }

**（2、FormattingConversionService：**继承自ConversionService，运行时类型转换和格式化服务接口，提供运行期类型转换和格式化的支持。

FormattingConversionService内部实现如下图所示：



我们可以看到FormattingConversionService内部实现如上所示，当你调用convert方法时：

⑴若是S类型----->String：调用私有的静态内部类PrinterConverter，其又调用相应的Printer的实现进行格式化；

⑵若是String----->T类型：调用私有的静态内部类ParserConverter，其又调用相应的Parser的实现进行解析；

⑶若是A注解类型注解的S类型----->String：调用私有的静态内部类AnnotationPrinterConverter，其又调用相应的AnnotationFormatterFactory的getPrinter获取Printer的实现进行格式化；

⑷若是String----->A注解类型注解的T类型：调用私有的静态内部类AnnotationParserConverter，其又调用相应的AnnotationFormatterFactory的getParser获取Parser的实现进行解析。

注：S类型表示源类型，T类型表示目标类型，A表示注解类型。

此处可以可以看出之前的Converter SPI完成任意Object与Object之间的类型转换，而Formatter SPI完成任意Object与String之间的类型转换（即格式化和解析，与PropertyEditor类似）。

**7.3.2、Spring内建的格式化转换器如下所示：**

|  |  |
| --- | --- |
| **类名** | **说明** |
| DateFormatter | java.util.Date<---->String  实现日期的格式化/解析 |
| NumberFormatter | java.lang.Number<---->String  实现通用样式的格式化/解析 |
| CurrencyFormatter | java.lang.BigDecimal<---->String  实现货币样式的格式化/解析 |
| PercentFormatter | java.lang.Number<---->String  实现百分数样式的格式化/解析 |
| NumberFormatAnnotationFormatterFactory | @NumberFormat注解类型的数字字段类型<---->String  ①通过@NumberFormat指定格式化/解析格式  ②可以格式化/解析的数字类型：Short、Integer、Long、Float、Double、BigDecimal、BigInteger |
| JodaDateTimeFormatAnnotationFormatterFactory | @DateTimeFormat注解类型的日期字段类型<---->String  ①通过@DateTimeFormat指定格式化/解析格式  ②可以格式化/解析的日期类型：  joda中的日期类型（org.joda.time包中的）：LocalDate、LocalDateTime、LocalTime、ReadableInstant  java内置的日期类型：Date、Calendar、Long    classpath中必须有Joda-Time类库，否则无法格式化日期类型 |

NumberFormatAnnotationFormatterFactory和JodaDateTimeFormatAnnotationFormatterFactory（如果classpath提供了Joda-Time类库）在使用格式化服务实现DefaultFormattingConversionService时会自动注册。

**7.3.3、示例**

在示例之前，我们需要到<http://joda-time.sourceforge.net/>下载Joda-Time类库，本书使用的是joda-time-2.1版本，将如下jar包添加到classpath：

**Java代码  [收藏代码](javascript:void())**

1. joda-time-2.1.jar

**7.3.3.1、类型级别的解析/格式化**

**一、直接使用Formatter SPI进行解析/格式化**

**Java代码  [收藏代码](javascript:void())**

1. //二、CurrencyFormatter：实现货币样式的格式化/解析
2. CurrencyFormatter currencyFormatter = **new** CurrencyFormatter();
3. currencyFormatter.setFractionDigits(2);//保留小数点后几位
4. currencyFormatter.setRoundingMode(RoundingMode.CEILING);//舍入模式（ceilling表示四舍五入）
6. //1、将带货币符号的字符串“$123.125”转换为BigDecimal("123.00")
7. Assert.assertEquals(**new** BigDecimal("123.13"), currencyFormatter.parse("$123.125", Locale.US));
8. //2、将BigDecimal("123")格式化为字符串“$123.00”展示
9. Assert.assertEquals("$123.00", currencyFormatter.print(**new** BigDecimal("123"), Locale.US));
10. Assert.assertEquals("￥123.00", currencyFormatter.print(**new** BigDecimal("123"), Locale.CHINA));
11. Assert.assertEquals("￥123.00", currencyFormatter.print(**new** BigDecimal("123"), Locale.JAPAN));

parse方法：将带格式的字符串根据Locale信息解析为相应的BigDecimal类型数据；

print方法：将BigDecimal类型数据根据Locale信息格式化为字符串数据进行展示。

不同于Convert SPI，Formatter SPI可以根据本地化（Locale）信息进行解析/格式化。

其他测试用例请参考cn.javass.chapter7.web.controller.support.formatter.InnerFormatterTest的testNumber测试方法和testDate测试方法。

**Java代码  [收藏代码](javascript:void())**

1. @Test
2. **public** **void** testWithDefaultFormattingConversionService() {
3. DefaultFormattingConversionService conversionService = **new** DefaultFormattingConversionService();
4. //默认不自动注册任何Formatter
5. CurrencyFormatter currencyFormatter = **new** CurrencyFormatter();
6. currencyFormatter.setFractionDigits(2);//保留小数点后几位
7. currencyFormatter.setRoundingMode(RoundingMode.CEILING);//舍入模式（ceilling表示四舍五入）
8. //注册Formatter SPI实现
9. conversionService.addFormatter(currencyFormatter);
11. //绑定Locale信息到ThreadLocal
12. //FormattingConversionService内部自动获取作为Locale信息，如果不设值默认是 Locale.getDefault()
13. LocaleContextHolder.setLocale(Locale.US);
14. Assert.assertEquals("$1,234.13", conversionService.convert(**new** BigDecimal("1234.128"), String.**class**));
15. LocaleContextHolder.setLocale(**null**);
17. LocaleContextHolder.setLocale(Locale.CHINA);
18. Assert.assertEquals("￥1,234.13", conversionService.convert(**new** BigDecimal("1234.128"), String.**class**));
19. Assert.assertEquals(**new** BigDecimal("1234.13"), conversionService.convert("￥1,234.13", BigDecimal.**class**));
20. LocaleContextHolder.setLocale(**null**);}

DefaultFormattingConversionService：带数据格式化功能的类型转换服务实现；

conversionService.addFormatter()：注册Formatter SPI实现；

conversionService.convert(**new** BigDecimal("1234.128"), String.**class**)：用于将BigDecimal类型数据格式化为字符串类型，此处根据“LocaleContextHolder.*setLocale*(locale)”设置的本地化信息进行格式化；

conversionService.convert("￥1,234.13", BigDecimal.**class**)：用于将字符串类型数据解析为BigDecimal类型数据，此处也是根据“LocaleContextHolder.*setLocale*(locale)”设置的本地化信息进行解；

LocaleContextHolder.*setLocale*(locale)：设置本地化信息到ThreadLocal，以便Formatter SPI根据本地化信息进行解析/格式化；

具体测试代码请参考cn.javass.chapter7.web.controller.support.formatter.InnerFormatterTest的testWithDefaultFormattingConversionService测试方法。

**三、自定义Formatter进行解析/格式化**

此处以解析/格式化PhoneNumberModel为例。

**（1、定义Formatter SPI实现**

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter7.web.controller.support.formatter;
2. //省略import
3. **public** **class** PhoneNumberFormatter **implements** Formatter<PhoneNumberModel> {
4. Pattern pattern = Pattern.compile("^(\\d{3,4})-(\\d{7,8})$");
5. @Override
6. **public** String print(PhoneNumberModel phoneNumber, Locale locale) {//①格式化
7. **if**(phoneNumber == **null**) {
8. **return** "";
9. }
10. **return** **new** StringBuilder().append(phoneNumber.getAreaCode()).append("-")
11. .append(phoneNumber.getPhoneNumber()).toString();
12. }
14. @Override
15. **public** PhoneNumberModel parse(String text, Locale locale) **throws** ParseException {//②解析
16. **if**(!StringUtils.hasLength(text)) {
17. //①如果source为空 返回null
18. **return** **null**;
19. }
20. Matcher matcher = pattern.matcher(text);
21. **if**(matcher.matches()) {
22. //②如果匹配 进行转换
23. PhoneNumberModel phoneNumber = **new** PhoneNumberModel();
24. phoneNumber.setAreaCode(matcher.group(1));
25. phoneNumber.setPhoneNumber(matcher.group(2));
26. **return** phoneNumber;
27. } **else** {
28. //③如果不匹配 转换失败
29. **throw** **new** IllegalArgumentException(String.format("类型转换失败，需要格式[010-12345678]，但格式是[%s]", text));
30. }
31. }
32. }

 类似于Convert SPI实现，只是此处的相应方法会传入Locale本地化信息，这样可以为不同地区进行解析/格式化数据。

**（2、测试用例：**

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter7.web.controller.support.formatter;
2. //省略import
3. **public** **class** CustomerFormatterTest {
4. @Test
5. **public** **void** test() {
6. DefaultFormattingConversionService conversionService = **new** DefaultFormattingConversionService();
7. conversionService.addFormatter(**new** PhoneNumberFormatter());
9. PhoneNumberModel phoneNumber = **new** PhoneNumberModel("010", "12345678");
10. Assert.assertEquals("010-12345678", conversionService.convert(phoneNumber, String.**class**));
12. Assert.assertEquals("010", conversionService.convert("010-12345678", PhoneNumberModel.**class**).getAreaCode());
13. }
14. }

通过PhoneNumberFormatter可以解析String--->PhoneNumberModel和格式化PhoneNumberModel--->String。

到此，类型级别的解析/格式化我们就介绍完了，从测试用例可以看出类型级别的是对项目中的整个类型实施相同的解析/格式化逻辑。

有的同学可能需要在不同的类的字段实施不同的解析/格式化逻辑，如用户模型类的注册日期字段只需要如“2012-05-02”格式进行解析/格式化即可，而订单模型类的下订单日期字段可能需要如“2012-05-02 20：13：13”格式进行展示。

接下来我们学习一下如何进行字段级别的解析/格式化吧。

**7.3.3.2、字段级别的解析/格式化**

**一、使用内置的注解进行字段级别的解析/格式化：**

**（1、测试模型类准备：**

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter7.model;
2. **public** **class** FormatterModel {
3. @NumberFormat(style=Style.NUMBER, pattern="#,###")
4. **private** **int** totalCount;
5. @NumberFormat(style=Style.PERCENT)
6. **private** **double** discount;
7. @NumberFormat(style=Style.CURRENCY)
8. **private** **double** sumMoney;
10. @DateTimeFormat(iso=ISO.DATE)
11. **private** Date registerDate;
13. @DateTimeFormat(pattern="yyyy-MM-dd HH:mm:ss")
14. **private** Date orderDate;
16. //省略getter/setter
17. }

此处我们使用了Spring字段级别解析/格式化的两个内置注解：

@Number：定义数字相关的解析/格式化元数据（通用样式、货币样式、百分数样式），参数如下：

    style：用于指定样式类型，包括三种：Style.NUMBER（通用样式） Style.CURRENCY（货币样式） Style.PERCENT（百分数样式），默认Style.NUMBER；

    pattern：自定义样式，如patter="#,###"；

@DateTimeFormat：定义日期相关的解析/格式化元数据，参数如下：

    pattern：指定解析/格式化字段数据的模式，如”yyyy-MM-dd HH:mm:ss”

iso：指定解析/格式化字段数据的ISO模式，包括四种：ISO.NONE（不使用）  ISO.DATE(yyyy-MM-dd) ISO.TIME(hh:mm:ss.SSSZ)  ISO.DATE\_TIME(yyyy-MM-dd hh:mm:ss.SSSZ)，默认ISO.NONE；

    style：指定用于格式化的样式模式，默认“SS”，具体使用请参考Joda-Time类库的org.joda.time.format.DateTimeFormat的forStyle的javadoc；

优先级： pattern 大于 iso 大于 style。

**（2、测试用例：**

**Java代码  [收藏代码](javascript:void())**

1. @Test
2. **public** **void** test() **throws** SecurityException, NoSuchFieldException {
3. //默认自动注册对@NumberFormat和@DateTimeFormat的支持
4. DefaultFormattingConversionService conversionService =
5. **new** DefaultFormattingConversionService();
7. //准备测试模型对象
8. FormatterModel model = **new** FormatterModel();
9. model.setTotalCount(10000);
10. model.setDiscount(0.51);
11. model.setSumMoney(10000.13);
12. model.setRegisterDate(**new** Date(2012-1900, 4, 1));
13. model.setOrderDate(**new** Date(2012-1900, 4, 1, 20, 18, 18));
15. //获取类型信息
16. TypeDescriptor descriptor =
17. **new** TypeDescriptor(FormatterModel.**class**.getDeclaredField("totalCount"));
18. TypeDescriptor stringDescriptor = TypeDescriptor.valueOf(String.**class**);
20. Assert.assertEquals("10,000", conversionService.convert(model.getTotalCount(), descriptor, stringDescriptor));
21. Assert.assertEquals(model.getTotalCount(), conversionService.convert("10,000", stringDescriptor, descriptor));
23. }

 TypeDescriptor：拥有类型信息的上下文，用于Spring3类型转换系统获取类型信息的（可以包含类、字段、方法参数、属性信息）；通过TypeDescriptor，我们就可以获取（类、字段、方法参数、属性）的各种信息，如注解类型信息；

conversionService.convert(model.getTotalCount(), descriptor, stringDescriptor)：将totalCount格式化为字符串类型，此处会根据totalCount字段的注解信息（通过descriptor对象获取）来进行格式化；

conversionService.convert("10,000", stringDescriptor, descriptor)：将字符串“10,000”解析为totalCount字段类型，此处会根据totalCount字段的注解信息（通过descriptor对象获取）来进行解析。

**（3、通过为不同的字段指定不同的注解信息进行字段级别的细粒度数据解析/格式化**

**Java代码  [收藏代码](javascript:void())**

1. descriptor = **new** TypeDescriptor(FormatterModel.**class**.getDeclaredField("registerDate"));
2. Assert.assertEquals("2012-05-01", conversionService.convert(model.getRegisterDate(), descriptor, stringDescriptor));
3. Assert.assertEquals(model.getRegisterDate(), conversionService.convert("2012-05-01", stringDescriptor, descriptor));
5. descriptor = **new** TypeDescriptor(FormatterModel.**class**.getDeclaredField("orderDate"));
6. Assert.assertEquals("2012-05-01 20:18:18", conversionService.convert(model.getOrderDate(), descriptor, stringDescriptor));
7. Assert.assertEquals(model.getOrderDate(), conversionService.convert("2012-05-01 20:18:18", stringDescriptor, descriptor));

 通过如上测试可以看出，我们可以通过字段注解方式实现细粒度的数据解析/格式化控制，但是必须使用TypeDescriptor来指定类型的上下文信息，即编程实现字段的数据解析/格式化比较麻烦。

其他测试用例请参考cn.javass.chapter7.web.controller.support.formatter.InnerFieldFormatterTest的test测试方法。

**二、自定义注解进行字段级别的解析/格式化：**

此处以解析/格式化PhoneNumberModel字段为例。

**（1、定义解析/格式化字段的注解类型：**

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter7.web.controller.support.formatter;
2. //省略import
3. @Target({ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER})
4. @Retention(RetentionPolicy.RUNTIME)
5. **public** **@interface** PhoneNumber {
6. }

**(2、实现AnnotationFormatterFactory注解格式化工厂：**

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter7.web.controller.support.formatter;
2. //省略import
3. **public** **class** PhoneNumberFormatAnnotationFormatterFactory
4. **implements** AnnotationFormatterFactory<PhoneNumber> {//①指定可以解析/格式化的字段注解类型
6. **private** **final** Set<Class<?>> fieldTypes;
7. **private** **final** PhoneNumberFormatter formatter;
8. **public** PhoneNumberFormatAnnotationFormatterFactory() {
9. Set<Class<?>> set = **new** HashSet<Class<?>>();
10. set.add(PhoneNumberModel.**class**);
11. **this**.fieldTypes = set;
12. **this**.formatter = **new** PhoneNumberFormatter();//此处使用之前定义的Formatter实现
13. }
14. //②指定可以被解析/格式化的字段类型集合
15. @Override
16. **public** Set<Class<?>> getFieldTypes() {
17. **return** fieldTypes;
18. }
19. //③根据注解信息和字段类型获取解析器
20. @Override
21. **public** Parser<?> getParser(PhoneNumber annotation, Class<?> fieldType) {
22. **return** formatter;
23. }
24. //④根据注解信息和字段类型获取格式化器
25. @Override
26. **public** Printer<?> getPrinter(PhoneNumber annotation, Class<?> fieldType) {
27. **return** formatter;
28. }
29. }

AnnotationFormatterFactory实现会根据注解信息和字段类型获取相应的解析器/格式化器。

**（3、修改FormatterModel添加如下代码：**

**Java代码  [收藏代码](javascript:void())**

1. @PhoneNumber
2. **private** PhoneNumberModel phoneNumber;

**（4、测试用例**

**Java代码  [收藏代码](javascript:void())**

1. @Test
2. ublic **void** test() **throws** SecurityException, NoSuchFieldException {
3. DefaultFormattingConversionService conversionService =
4. **new** DefaultFormattingConversionService();//创建格式化服务
5. conversionService.addFormatterForFieldAnnotation(
6. **new** PhoneNumberFormatAnnotationFormatterFactory());//添加自定义的注解格式化工厂
8. FormatterModel model = **new** FormatterModel();
9. TypeDescriptor descriptor =
10. **new** TypeDescriptor(FormatterModel.**class**.getDeclaredField("phoneNumber"));
11. TypeDescriptor stringDescriptor = TypeDescriptor.valueOf(String.**class**);
13. PhoneNumberModel value = (PhoneNumberModel) conversionService.convert("010-12345678", stringDescriptor, descriptor); //解析字符串"010-12345678"--> PhoneNumberModel
14. model.setPhoneNumber(value);
16. Assert.assertEquals("010-12345678", conversionService.convert(model.getPhoneNumber(), descriptor, stringDescriptor));//格式化PhoneNumberModel-->"010-12345678"

此处使用DefaultFormattingConversionService的addFormatterForFieldAnnotation注册自定义的注解格式化工厂PhoneNumberFormatAnnotationFormatterFactory。

到此，编程进行数据的格式化/解析我们就完成了，使用起来还是比较麻烦，接下来我们将其集成到Spring Web MVC环境中。

7.3.4、集成到Spring Web MVC环境

**一、注册FormattingConversionService实现和自定义格式化转换器：**

**Java代码  [收藏代码](javascript:void())**

1. <bean id="conversionService"
2. **class**="org.springframework.format.support.FormattingConversionServiceFactoryBean">
3. <!—此处省略之前注册的自定义类型转换器-->
4. <property name="formatters">
5. <list>
6. <bean **class**="cn.javass.chapter7.web.controller.support.formatter.
7. PhoneNumberFormatAnnotationFormatterFactory"/>
8. </list>
9. </property>
10. </bean>

 其他配置和之前学习7.2.2.4一节一样。

**二、示例：**

**（1、模型对象字段的数据解析/格式化：**

**Java代码  [收藏代码](javascript:void())**

1. @RequestMapping(value = "/format1")
2. **public** String test1(@ModelAttribute("model") FormatterModel formatModel) {
3. **return** "format/success";
4. }

**Java代码  [收藏代码](javascript:void())**

1. totalCount:<spring:bind path="model.totalCount">${status.value}</spring:bind><br/>
2. discount:<spring:bind path="model.discount">${status.value}</spring:bind><br/>
3. sumMoney:<spring:bind path="model.sumMoney">${status.value}</spring:bind><br/>
4. phoneNumber:<spring:bind path="model.phoneNumber">${status.value}</spring:bind><br/>
5. <!-- 如果没有配置org.springframework.web.servlet.handler.ConversionServiceExposingInterceptor将会报错 -->
6. phoneNumber:<spring:eval expression="model.phoneNumber"></spring:eval><br/>
8. <br/><br/>
9. <form:form commandName="model">
10. <form:input path="phoneNumber"/><br/>
11. <form:input path="sumMoney"/>
12. </form:form>

在浏览器输入测试URL：

http://localhost:9080/springmvc-chapter7/format1?totalCount=100000&discount=0.51&sumMoney=100000.128&phoneNumber=010-12345678

数据会正确绑定到我们的formatModel，即请求参数能被正确的解析并绑定到我们的命令对象上，而且在JSP页面也能正确的显示格式化后的数据（即正确的被格式化显示）。

**（2、功能处理方法参数级别的数据解析：**

**Java代码  [收藏代码](javascript:void())**

1. @RequestMapping(value = "/format2")
2. **public** String test2(
3. @PhoneNumber @RequestParam("phoneNumber") PhoneNumberModel phoneNumber,
4. @DateTimeFormat(pattern="yyyy-MM-dd") @RequestParam("date") Date date) {
5. System.out.println(phoneNumber);
6. System.out.println(date);
7. **return** "format/success2";
8. }

此处我们可以直接在功能处理方法的参数上使用格式化注解类型进行注解，Spring Web MVC能根据此注解信息对请求参数进行解析并正确的绑定。

在浏览器输入测试URL：

http://localhost:9080/springmvc-chapter7/format2?phoneNumber=010-12345678&date=2012-05-01

数据会正确的绑定到我们的phoneNumber和date上，即请求的参数能被正确的解析并绑定到我们的参数上。

控制器代码位于cn.javass.chapter7.web.controller.DataFormatTestController中。

如果我们请求参数数据不能被正确解析并绑定或输入的数据不合法等该怎么处理呢？接下来的一节我们来学习下绑定失败处理和数据验证相关知识。

**7.4、数据验证**

**7.4.1、编程式数据验证**

Spring 2.x提供了编程式验证支持，详见【4.16.2 数据验证】章节，在此我们重写【4.16.2.4.1、编程式验证器】一节示例。

**（1、验证器实现**

复制cn.javass.chapter4.web.controller.support.validator.UserModelValidator

到cn.javass.chapter7.web.controller.support.validator.UserModelValidator。

**（2、控制器实现**

**Java代码  [收藏代码](javascript:void())**

1. @Controller
2. **public** **class** RegisterSimpleFormController {
3. **private** UserModelValidator validator = **new** UserModelValidator();
5. @ModelAttribute("user")        //① 暴露表单引用对象为模型数据
6. **public** UserModel getUser() {
7. **return** **new** UserModel();
8. }
9. @RequestMapping(value = "/validator", method = RequestMethod.GET)
10. **public** String showRegisterForm() {   //② 表单展示
11. **return** "validate/registerAndValidator";
12. }
13. @RequestMapping(value = "/validator", method = RequestMethod.POST)
14. **public** String submitForm(
15. @ModelAttribute("user") UserModel user,
16. Errors errors) {           //③ 表单提交
17. validator.validate(user, errors);  //1 调用UserModelValidator的validate方法进行验证
18. **if**(errors.hasErrors()) { //2如果有错误再回到表单展示页面
19. **return** showRegisterForm();
20. }
21. **return** "redirect:/success";
22. }
23. }

在submitForm方法中，我们首先调用之前写的UserModelValidator的validate方法进行验证，当然此处可以直接验证并通过Errors接口来保留错误；此处还通过 Errors接口的hasErrors方法来决定当验证失败时显示的错误页面。

**（3、spring配置文件chapter7-servlet.xml**

**Java代码  [收藏代码](javascript:void())**

1. <bean **class**="cn.javass.chapter7.web.controller.RegisterSimpleFormController"/>

**（4、错误码配置（messages.properties），需要执行NativeToAscii**

直接将【springmvc-chapter4】项目中src下的messages.properties复制到src目录下。

在spring配置文件chapter7-servlet.xml中添加messageSource：

**Java代码  [收藏代码](javascript:void())**

1. <bean id="messageSource"
2. **class**="org.springframework.context.support.ReloadableResourceBundleMessageSource">
3. <property name="basename" value="classpath:messages"/>
4. <property name="fileEncodings" value="utf-8"/>
5. <property name="cacheSeconds" value="120"/>
6. </bean>

**（5、视图页面（/WEB-INF/jsp/registerAndValidator.jsp）**

直接将【springmvc-chapter4】项目中的/WEB-INF/jsp/registerAndValidator.jsp复制到当前项目下的/WEB-INF/jsp/validate/registerAndValidator.jsp。

**（6、启动服务器测试：**

在浏览器地址栏输入http://localhost:9080/springmvc-chapter7/validator进行测试，测试步骤和【4.16.2.4.1、编程式验证器】一样。

其他编程式验证的使用，请参考【4.16.2 数据验证】章节。

**7.4.2、声明式数据验证**

Spring3开始支持JSR-303验证框架，JSR-303支持XML风格的和注解风格的验证，接下来我们首先看一下如何和Spring集成。

7.4.2.1、集成

**（1、添加jar包：**

此处使用Hibernate-validator实现（版本：hibernate-validator-4.3.0.Final-dist.zip），将如下jar包添加到classpath（WEB-INF/lib下即可）：

**写道**

dist/lib/required/validation-api-1.0.0.GA.jar JSR-303规范API包  
dist/hibernate-validator-4.3.0.Final.jar Hibernate 参考实现

**（2、在Spring配置总添加对JSR-303验证框架的支持**

**Java代码  [收藏代码](javascript:void())**

1. <!-- 以下 validator  ConversionService 在使用 mvc:annotation-driven 会 自动注册-->
2. <bean id="validator"
3. **class**="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
4. <property name="providerClass"  value="org.hibernate.validator.HibernateValidator"/>
5. <!-- 如果不加默认到 使用classpath下的 ValidationMessages.properties -->
6. <property name="validationMessageSource" ref="messageSource"/>
7. </bean>

此处使用Hibernate validator实现：

validationMessageSource属性：指定国际化错误消息从哪里取，此处使用之前定义的messageSource来获取国际化消息；如果此处不指定该属性，则默认到classpath下的ValidationMessages.properties取国际化错误消息。

通过ConfigurableWebBindingInitializer注册validator：

**Java代码  [收藏代码](javascript:void())**

1. <bean id="webBindingInitializer"
2. **class**="org.springframework.web.bind.support.ConfigurableWebBindingInitializer">
3. <property name="conversionService" ref="conversionService"/>
4. <property name="validator" ref="validator"/>
5. </bean>

其他配置和之前学习7.2.2.4一节一样。

如上集成过程看起来比较麻烦，后边我们会介绍<mvc:annotation-driven>和@EnableWebMvc，ConversionService会自动注册，后续章节再详细介绍。

**（3、使用JSR-303验证框架注解为模型对象指定验证信息**

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter7.model;
2. **import** javax.validation.constraints.NotNull;
3. **public** **class** UserModel {
4. @NotNull(message="{username.not.empty}")
5. **private** String username;
6. }

通过@NotNull指定此username字段不允许为空，当验证失败时将从之前指定的messageSource中获取“username.not.empty”对于的错误信息，此处只有通过“{错误消息键值}”格式指定的才能从messageSource获取。

**（4、控制器**

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter7.web.controller.validate;
2. //省略import
3. @Controller
4. **public** **class** HelloWorldController {
5. @RequestMapping("/validate/hello")
6. **public** String validate(@Valid @ModelAttribute("user") UserModel user, Errors errors) {
8. **if**(errors.hasErrors()) {
9. **return** "validate/error";
10. }
11. **return** "redirect:/success";
12. }
13. }

通过在命令对象上注解@Valid来告诉Spring MVC此命令对象在绑定完毕后需要进行JSR-303验证，如果验证失败会将错误信息添加到errors错误对象中。

**（5、验证失败后需要展示的页面（/WEB-INF/jsp/validate/error.jsp）**

**Java代码  [收藏代码](javascript:void())**

1. <%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
2. <%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
4. <form:form commandName="user">
5. <form:errors path="\*" cssStyle="color:red"></form:errors><br/>
6. </form:form>

**（6、测试**

在浏览器地址栏中输入<http://localhost:9080/springmvc-chapter7/validate/hello>，即没有username数据，请求后将直接到验证失败界面并显示错误消息“用户名不能为空”，如果请求时带上“?username=zhang”将重定向到成功页面。

到此集成就完成，接下来我们详细学习下有哪些验证约束注解吧。

7.4.2.2、内置的验证约束注解

内置的验证约束注解如下表所示（摘自hibernate validator reference）：

|  |  |  |
| --- | --- | --- |
| **验证注解** | **验证的数据类型** | **说明** |
| @AssertFalse | Boolean,boolean | 验证注解的元素值是false |
| @AssertTrue | Boolean,boolean | 验证注解的元素值是true |
| @NotNull | 任意类型 | 验证注解的元素值不是null |
| @Null | 任意类型 | 验证注解的元素值是null |
| @Min(value=值) | BigDecimal，BigInteger, byte,  short, int, long，等任何Number或CharSequence（存储的是数字）子类型 | 验证注解的元素值大于等于@Min指定的value值 |
| @Max（value=值） | 和@Min要求一样 | 验证注解的元素值小于等于@Max指定的value值 |
| @DecimalMin(value=值) | 和@Min要求一样 | 验证注解的元素值大于等于@ DecimalMin指定的value值 |
| @DecimalMax(value=值) | 和@Min要求一样 | 验证注解的元素值小于等于@ DecimalMax指定的value值 |
| @Digits(integer=整数位数, fraction=小数位数) | 和@Min要求一样 | 验证注解的元素值的整数位数和小数位数上限 |
| @Size(min=下限, max=上限) | 字符串、Collection、Map、数组等 | 验证注解的元素值的在min和max（包含）指定区间之内，如字符长度、集合大小 |
| @Past | java.util.Date,  java.util.Calendar;  Joda Time类库的日期类型 | 验证注解的元素值（日期类型）比当前时间早 |
| @Future | 与@Past要求一样 | 验证注解的元素值（日期类型）比当前时间晚 |
| @NotBlank | CharSequence子类型 | 验证注解的元素值不为空（不为null、去除首位空格后长度为0），不同于@NotEmpty，@NotBlank只应用于字符串且在比较时会去除字符串的首位空格 |
| @Length(min=下限, max=上限) | CharSequence子类型 | 验证注解的元素值长度在min和max区间内 |
| @NotEmpty | CharSequence子类型、Collection、Map、数组 | 验证注解的元素值不为null且不为空（字符串长度不为0、集合大小不为0） |
| @Range(min=最小值, max=最大值) | BigDecimal,BigInteger,CharSequence, byte, short, int, long等原子类型和包装类型 | 验证注解的元素值在最小值和最大值之间 |
| @Email(regexp=正则表达式,  flag=标志的模式) | CharSequence子类型（如String） | 验证注解的元素值是Email，也可以通过regexp和flag指定自定义的email格式 |
| @Pattern(regexp=正则表达式,  flag=标志的模式) | String，任何CharSequence的子类型 | 验证注解的元素值与指定的正则表达式匹配 |
| @Valid | 任何非原子类型 | 指定递归验证关联的对象；  如用户对象中有个地址对象属性，如果想在验证用户对象时一起验证地址对象的话，在地址对象上加@Valid注解即可级联验证 |

此处只列出Hibernate Validator提供的大部分验证约束注解，请参考hibernate validator官方文档了解其他验证约束注解和进行自定义的验证约束注解定义。

具体演示实例请参考cn.javass.chapter7.web.controller.validate.ValidatorAnnotationTestController。

7.4.2.3、错误消息

当验证出错时，我们需要给用户展示错误消息告诉用户出错的原因，因此我们要为验证约束注解指定错误消息。错误消息是通过在验证约束注解的message属性指定。验证约束注解指定错误消息有如下两种方式：

1、硬编码错误消息；

2、从资源消息文件中根据消息键读取错误消息。

**一、硬编码错误消息**

直接在验证约束注解上指定错误消息，如下所示：

**Java代码  [收藏代码](javascript:void())**

1. @NotNull(message = "用户名不能为空")
2. @Length(min=5, max=20, message="用户名长度必须在5-20之间")
3. @Pattern(regexp = "^[a-zA-Z\_]\\w{4,19}$", message = "用户名必须以字母下划线开头，可由字母数字下划线组成")
4. **private** String username;

如上所示，错误消息使用硬编码指定，这种方式是不推荐使用的，因为在如下场景是不适用的：

1、在国际化场景下，需要对不同的国家显示不同的错误消息；

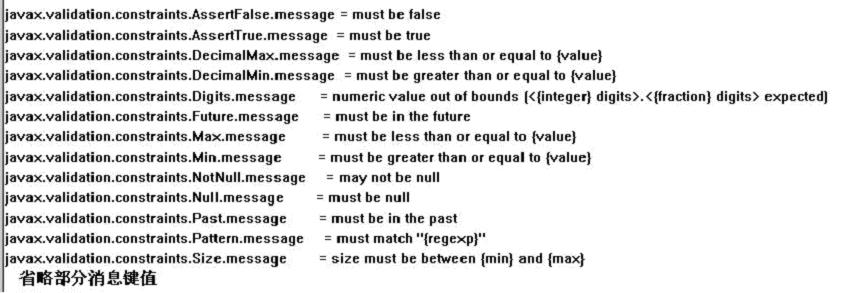
2、需要更换错误消息时是比较麻烦的，需要找到相应的类进行更换，并重新编译发布。

**二、从资源消息文件中根据消息键读取错误消息**

**2.1、默认的错误消息文件及默认错误消息键值**

默认的错误消息文件是/org/hibernate/validator/ValidationMessages.properties，如下图所示：

  
 默认的错误消息键值如下图所示：

  
 消息键默认为：验证约束注解的全限定类名.message

在我们之前的测试文件中，错误消息键值是使用默认的，如何自定义错误消息文件和错误消息键值呢？

**2.2、自定义的错误消息文件和错误消息键值**

自定义的错误消息文件里的错误消息键值将覆盖默认的错误消息文件中的错误消息键值。我们自定义的错误消息文件是具有国际化功能的。

**（1、定义错误消息文件**

在类装载路径的根下创建ValidationMessages.properties文件，如在src目录下创建会自动复制到类装载路径的根下，并添加如下消息键值（需要native2ascii，可以在eclipse里装Properties Editor，自动保存为ASCII码）：

**Java代码  [收藏代码](javascript:void())**

1. javax.validation.constraints.Pattern.message=用户名必须以字母或下划线开头，后边可以跟字母数字下划线，长度在5-20之间

需要在你的spring配置文件WEB-INF/chapter7-servlet.xml修改之前的validator Bean：

**Java代码  [收藏代码](javascript:void())**

1. <bean id="validator"
2. **class**="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean">
3. <property name="providerClass"
4. value="org.hibernate.validator.HibernateValidator"/>
5. </bean>

此时错误消息键值的查找会先到classpath下ValidationMessages.properties中找，找不到再到默认的错误消息文件中找。

输入测试地址：http://localhost:9080/springmvc-chapter7/validate/pattern?value=zhan，将看到我们自定义的错误消息显示出来了。

**（2、使用Spring的MessageSource Bean进行消息键值的查找**

如果我们的环境是与spring集成，还是应该使用Spring提供的消息支持，具体配置如下：

在spring配置文件WEB-INF/chapter7-servlet.xml定义MessageSource Bean：

**Java代码  [收藏代码](javascript:void())**

1. <bean id="messageSource"
2. **class**="org.springframework.context.support.ReloadableResourceBundleMessageSource">
3. <property name="basename" value="classpath:messages"/>
4. <property name="fileEncodings" value="utf-8"/>
5. <property name="cacheSeconds" value="120"/>
6. </bean>

之前我们已经配置过了，在此就不详述了。

在spring配置文件WEB-INF/chapter7-servlet.xml定义的validator Bean，添加如下属性：

**Java代码  [收藏代码](javascript:void())**

1. <property name="validationMessageSource" ref="messageSource"/>

 验证失败的错误消息键值的查找将使用messageSource Bean进行。

在消息文件src/messages.properties中添加如下错误消息：

**Java代码  [收藏代码](javascript:void())**

1. javax.validation.constraints.Pattern.message=用户名必须以字母或下划线开头，后边可以跟字母数字下划线，长度在5-20之间

输入测试地址：http://localhost:9080/springmvc-chapter7/validate/pattern?value=zhan，将看到我们自定义的错误消息显示出来了。

**当我们配置了messageSource Bean时，默认将为验证的对象自动生成如下错误消息键：**

验证错误注解简单类名.验证对象名.字段名

验证错误注解简单类名.字段名

验证错误注解简单类名.字段类型全限定类名

验证错误注解简单类名

使用的优先级是：从高到低，即最前边的具有最高的优先级，而且以上所有默认的错误消息键优先级高于自定义的错误消息键。

如测试用例cn.javass.chapter7.web.controller.validate.ValidatorAnnotationTestController中的public String pattern(@Valid @ModelAttribute("model") PatternModel model, Errors errors)将自动产生如下错误消息键：

Pattern.model.value=验证错误注解简单类名.验证对象名.字段名

Pattern.value=验证错误注解简单类名.字段名

Pattern.java.lang.String=验证错误注解简单类名.字段类型全限定类名

Pattern=验证错误注解简单类名

**（3、自定义错误消息键值**

之前我们已经学习了硬编码错误消息，及默认的错误消息，在大部分场景下，以上两种方式无法满足我们的需求，因此我们需要自定义错误消息键值。

在验证约束注解上指定错误消息键：

**Java代码  [收藏代码](javascript:void())**

1. **package** cn.javass.chapter7.web.controller.validate.model;
2. **public** **class** PatternModel {
3. @Pattern(regexp = "^[a-zA-Z\_][\\w]{4,19}$", message="{user.name.error}")
4. **private** String value;
5. }

我们可以通过验证约束注解的message属性指定错误消息键，格式如“{消息键}”。

在消息文件src/messages.properties中添加如下错误消息：

**Java代码  [收藏代码](javascript:void())**

1. user.name.error=用户名格式不合法

输入测试地址：http://localhost:9080/springmvc-chapter7/validate/pattern?value=zhan，将看到我们自定义的错误消息显示出来了。

接下来我们看下如下场景

**Java代码  [收藏代码](javascript:void())**

1. @Length(min=5, max=20, message="{user.name.length.error}")

**Java代码  [收藏代码](javascript:void())**

1. user.name.error=用户名长度必须在5-20之间

错误消息中的5-20应该是从@Length验证约束注解中获取的，而不是在错误消息中硬编码，因此我们需要占位符的支持：

●如@Length(min=5, max=20, message="{user.name.length.error}")，错误消息可以这样写：用户名长度必须在{min}-{max}之间

**错误消息占位符规则：**

{验证注解属性名}，如@Length有min和max属性，则在错误消息文件中可以通过{min}和{max}来获取；如@Max有value属性，则在错误消息文件中可以通过{value}来获取。

**Java代码  [收藏代码](javascript:void())**

1. user.name.length.error=用户名长度必须在{min}-{max}之间

输入测试地址：http://localhost:9080/springmvc-chapter7/validate/length?value=1，将看到我们自定义的错误消息显示出来了。

7.4.2.4、功能处理方法上多个验证参数的处理

当我们在一个功能处理方法上需要验证多个模型对象时，需要通过如下形式来获取验证结果：

**Java代码  [收藏代码](javascript:void())**

1. @RequestMapping("/validate/multi")
2. **public** String multi(
3. @Valid @ModelAttribute("a") A a, BindingResult aErrors,
4. @Valid @ModelAttribute("b") B b, BindingResult bErrors) {
6. **if**(aErrors.hasErrors()) { //如果a模型对象验证失败
7. **return** "validate/error";
8. }
9. **if**(bErrors.hasErrors()) { //如果a模型对象验证失败
10. **return** "validate/error";
11. }
12. **return** "redirect:/success";
13. }

每一个模型对象后边都需要跟一个Errors或BindingResult对象来保存验证结果，其方法体内部可以使用这两个验证结果对象来选择出错时跳转的页面。详见cn.javass.chapter7.web.controller.validate.MultiModelController。

在错误页面，需要针对不同的模型来显示错误消息：

**Java代码  [收藏代码](javascript:void())**

1. <form:form commandName="a">
2. <form:errors path="\*" cssStyle="color:red"></form:errors><br/>
3. </form:form>
4. <form:form commandName="b">
5. <form:errors path="\*" cssStyle="color:red"></form:errors><br/>
6. </form:form>